Lecture 2: from SOAP towards REST

TIES4560 SOA and Cloud Computing
Autumn 2023

University of Jyväskylä

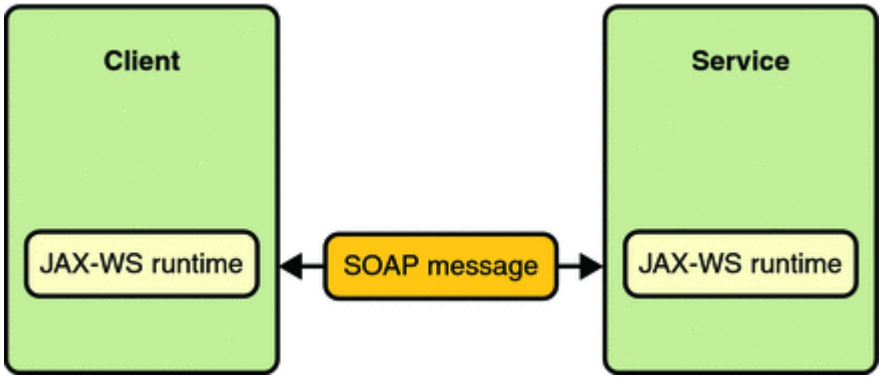
Khriyenko Oleksiy

SOAP Web Services

JAX-WS

JAX-WS *is a set of APIs to build SOAP Web Services. JAX-WS provides many annotation to simplify the development and deployment for both web service clients and web service providers (endpoints).*

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.



Related tutorials:

- *http://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html*
- *http://www.mkyong.com/tutorials/jax-ws-tutorials/*
- *http://www.mkyong.com/webservices/jax-ws/jax-ws-hello-world-example/*
- *https://www.baeldung.com/jax-ws*
- *https://axis.apache.org/axis2/java/core/docs/jaxws-guide.html*
- *https://spring.io/guides/gs/producing-web-service/*
- *https://spring.io/guides/gs/consuming-web-service/*
- *https://dzone.com/articles/creating-a-soap-web-service-with-spring-boot-start*

Dynamic Client to a SOAP Web Service

The JAX-WS supports both the ***dynamic*** and ***static*** client programming models that enable both synchronous and asynchronous invocation of JAX-WS web services.

The ***Dispatch client API*** is a dynamic client programming model for JAX-WS. The Dispatch interface provides support for the dynamic invocation of a service endpoint operations. SOAP web services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The Dispatch client API, *javax.xml.ws.Dispatch*, is an XML messaging-oriented client that is intended for advanced XML developers who prefer using XML constructs. You do not need a WSDL file if you are developing a dynamic client.

Related tutorials:

- *http://www.ibm.com/support/knowledgecenter/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/twbs_devwbsjaxwsclient_dyn.html*
- *https://www.ibm.com/support/knowledgecenter/SS7K4U_8.5.5/com.ibm.websphere.base.doc/ae/twbs_jaxwsdynclient.html*
- *http://cxf.apache.org/docs/jax-ws-dispatch-api.html*
- *http://cxf.apache.org/docs/how-do-i-develop-a-client.html*
- *http://docs.oracle.com/cd/E21764_01/web.1111/e13734/provider.htm#WSADV583*
- *http://java.boot.by/ocewsd6-guide/ch10s04.html*
- *http://www.programcreek.com/java-api-examples/index.php?api=javax.xml.ws.Dispatch*

Dynamic Client to a SOAP Web Service

The ***Dynamic proxy client API*** is a static client programming model for JAX-WS. In contrast to Dispatch client, the Dynamic proxy client invokes a web service based on a service endpoint interface (SEI) that is generated from a WSDL file. If you do not want to work directly with XML (work with either the message structure or the message payload structure) and do appreciate work with a Java abstraction and want the Web services client to invoke the service based on service endpoint interfaces with a dynamic proxy, use the Dynamic Proxy API to develop a static web service client. The Dynamic Proxy client is similar to the ***Static proxy client*** (stub client) in the Java API for XML-based RPC (JAX-RPC) programming model.

Although the JAX-WS Dynamic Proxy client and the Static proxy client (JAX-RPC stub client) are both based on the Service Endpoint Interface (SEI) that is generated from a WSDL file, there is a major difference. Static proxy client compile and bind the Web service client at development time. This generates a static stub for the Web service client proxy. The source code for the generated static stub client relies on a specific service implementation. As a result, this option offers the least flexibility. Unlike the JAX-RPC stub clients, the Dynamic Proxy client does not require you to regenerate a stub prior to running the client on an application server for a different vendor because the generated interface does not require the specific vendor information. The Dynamic Proxy client is dynamically generated at run time using the Java Dynamic Proxy functionality. This option does not rely upon a specific service implementation, providing greater flexibility, but also a greater performance hit.

The Generated Client classes are great if you know precisely what web-service your client code is going to call and that it isn't going to change over the lifetime of your client. In case of Dynamic Client, you don't need to have generated stubs before run time. This allows you to generically invoke services that you may not know about beforehand. Comparing to Generated Stub(GS) it works slower, because of run-time stub generation.

Related tutorials:

- *http://www.xyzws.com/scdjws/SGS34/6*
- *http://docs.oracle.com/cd/E23943_01/web.1111/e13734/proxy.htm#WSADV146*
- *https://docs.oracle.com/middleware/1213/wls/WSRPC/jax-rpc-client.htm#WSRPC199*

Dynamic Client to a SOAP Web Service

The ***Dynamic Invocation Interface (DII)*** client does not require a WSDL file to generate static stubs or pass the WSDL file to the service factory to create the service; instead, the client must know a service's address, operations, and parameters in advance. Using the Dynamic Invocation Interface (DII) enables the client to discover target services dynamically on runtime and then to invoke methods. During runtime, the client uses a set of service operations and parameters, establishes a search criterion to discover the target service, and then invokes its methods. This also enables a DII client to invoke a service and its methods without knowing its data types, objects, and its return types.

With the *Dynamic Invocation Interface (DII),* a client can call a remote procedure even if the signature of the remote procedure or the name of the service are unknown until runtime. Because of its flexibility, a DII client can be used in a service broker that dynamically discovers services, configures the remote calls, and executes the calls.

Related tutorials:

- *http://www.xyzws.com/scdjws/SGS34/6*
- *https://docs.oracle.com/cd/E17802_01/j2ee/j2ee/1.4/docs/tutorial-update2/doc/JAXRPC5.html*
- *http://www.inf.fu-berlin.de/lehre/SS03/19560-P/Docs/JWSDP/tutorial/doc/JAXRPC6.html*

Demo

Simple SOAP Web Service

- Create a simple Java project. (In case of missing http server issues, add corresponding libraries or make Dynamic Web Project)
- Define a simple Interface that will represent a Web Service

```
…
public interface PService {
            public int add(int a, int b);
}
```

- Create a service implementation class that implements the Interface of Web Service and annotate it with *@WebService* annotation

```
import javax.jws.WebService;
@WebService
public class PServiceImpl implements PService {
            @Override
            @WebMethod
            public int add(int a, int b) { return a+b;}
}
```

Some Web Service annotations could be also applied to the Interface definition. And this is actually recommended! In this case, actual implementation class should be annotated with specification of endpointInterface attribute that refers to the interface class (incl. package name).

- Create a publisher for the Web Service. Specify a Web Service URL and the class of service implementation.

```
import javax.xml.ws.Endpoint;

public class Exporter {
            public static void main(String[] args) {
                        Endpoint.publish("http://localhost:8080/MyServices/pservice", new PServiceImpl());
            }
}
```

Publisher will work only if your implementation of jax-ws supports endpoint publishing. As long as you are using Sun virtual machine or Oracle JDK, that would be fine.

- Run publisher as a java application and check the URL of the published endpoint of the Web Service. There you will find basic information of the published Web Service as well as its WSDL.

Demo

Simple SOAP Web Service

- Some dependences that might be useful…

```
…
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>

<dependency>
    <groupId>javax.xml.ws</groupId>
    <artifactId>jaxws-api</artifactId>
    <version>2.3.1</version>
</dependency>

<dependency>
    <groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-rt</artifactId>
    <version>2.3.1</version>
</dependency>

<dependency>
    <groupId>com.sun.net.httpserver</groupId>
    <artifactId>http</artifactId>
    <version>20070405</version>
</dependency>
```

JDK16

```
…
<dependencies>
  <dependency>
    <groupId>jakarta.xml.ws</groupId>
    <artifactId>jakarta.xml.ws-api</artifactId>
    <version>4.0.0</version>
  </dependency>
  <dependency>
    <groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-rt</artifactId>
    <version>4.0.0</version>
  </dependency>
</dependencies>
```

The biggest change in Java 11 was the removal of the Java EE and CORBA modules such as the four web services APIs - JAX-WS, JAXB, JAF and Common Annotations - that were deemed redundant since they were already included in Java EE. Oracle donated Java EE 8 to the Eclipse Foundation shortly after its release in 2017 with the intent that Java EE be open-sourced. Due to Oracle's branding policy, it was necessary to rename Java EE to Jakarta EE and migrate the namespace from **javax** to **jakarta .**

Related tutorials:

- *https://www.infoq.com/articles/why-how-upgrade-java17/*

Demo

Simple SOAP Web Service with Tomcat

- Create a Java Dynamic Web Project
- Create a class that will represent a Web Service functionality

```
…
@WebService (targetNamespace="http://myWSService")
public class PService {
            @WebMethod
            public int add(int a, int b) { return a+b;}
}
```

- Create a Web Service (Fine > New > Other… > Web Services > Web Service) based on created class.

Tomcat will publish the Web Service under temporal URL , as well as, automatically generated WSDL file.

Corresponding human friendly Client for the service will be automatically generated to test functionality.

You might need to specify:

- Apache CXF directory (binary distribution with jar files) in Eclipse (Window>Preferences>Web Services>CXF Preferences)
- TargetNamespace (that might refer to the location of you service class in the project)

- To generate a web service client, create a project and run wizard (Fine > New > Other… > Web Services > Web Service Client) specifying location of WSDL file of the service. Configure automatic generation of a Client (level of readiness <Test client>).
- WSDL is accessible following the endpoint of the Web Service (e.g. *http://localhost:6173/PWS_web/services/PService?wsdl*)

Having WSDL description of the Web Service you may build a Client using generated java code by wsimport tool.

- *Web Service Explorer* – imbedded as a plug-in into Eclipse.
- *SoapUI* - External Web Service testing tool (*https://www.soapui.org/*). It is also available as a plug-in for Eclipse.

Guide: *http://www2.smartbear.com/rs/smartbear/images/SmartBear-SoapUI-101-eBook.pdf*

- Other Web Service testing tools: *https://www.guru99.com/top-6-api-testing-tool.html*

SOAP Web Service extra (WSDL customization)

- *Service First* vs. *Contract First* strategies

```
package org.ws.myservise;
import javax.jws.WebMethod;
import javax.jws.WebService;

//@WebService (name="name",
         portName="portName",
         serviceName="serviceName",
         targetNamespace="targetNamespace",
         endpointInterface="endpointInterface",
         wsdlLocation="wsdlLocation" )
//@SOAPBinding (style=Style.RPC or .DOCUMENT,
         use="…")

public class ServiceLogic {

//@WebMethod (operationName="operationName",
            action="action",
            exclude=true or false)
//@WebResult (partName="partName")
   public String helloName(@WebParam
(partName="partName")String name){
      return "Hello there " + name;
   }
}
```

There are also some other annotations to configure *Input* and *Output* types (*@SOAPBinding*, *@WebParam*, *@WebResult*)

JAXB annotation (*@XmlRootElement*, *@XmlType*, *@XmlElement*) is used to customize XML based custom types definitions

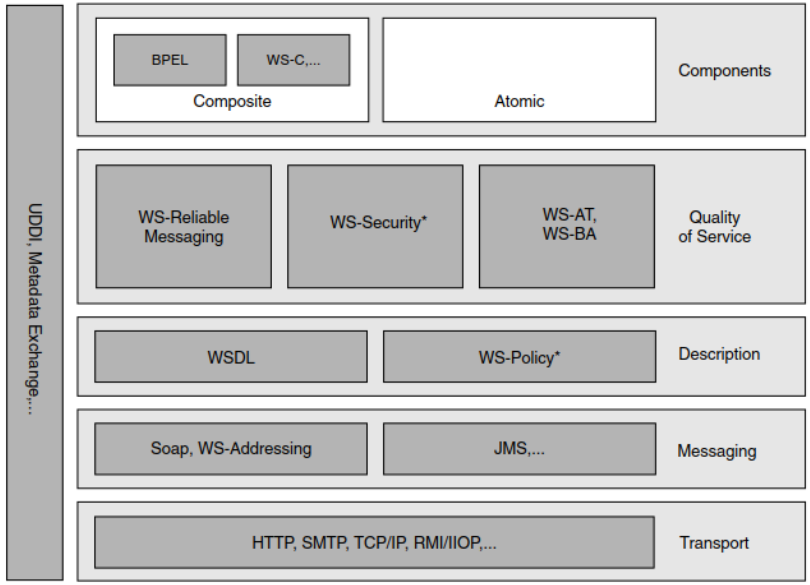
```
…
<wsdl:definitions targetNamespace="http://myservice.ws.org"  …>
…
<wsdl:message name="helloNameResponse">
    <wsdl:part element="impl:helloNameResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="helloNameRequest">
    <wsdl:part element="impl:helloName" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="ServiceLogic">
    <wsdl:operation name="helloName">
      <wsdl:input message="impl:helloNameRequest" name="helloNameRequest">
     </wsdl:input>
      <wsdl:output message="impl:helloNameResponse"
                   name="helloNameResponse">
     </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="ServiceLogicSoapBinding" type="impl:ServiceLogic">
    <wsdlsoap:binding style="document"
                      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="helloName">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="helloNameRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="helloNameResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ServiceLogicService">
    <wsdl:port binding="impl:ServiceLogicSoapBinding" name="ServiceLogic">
      <wsdlsoap:address
                location="http://localhost:8080/MyWS/services/ServiceLogic"/>
    </wsdl:port>
  </wsdl:service>
```

From SOAP towards REST

Overall, ***Web Services architecture*** consists of many layers including protocols and standards for security and reliability. Therefore, it becomes complicated and time/resource consuming for developers of simple web services.

Web Services can be implemented in Web environments too, on top of basic Web technologies such as HTTP, Simple Mail Transfer Protocol (SMTP), and so on.

REpresentational State Transfer (REST) has gained widespread acceptance across the Web as a simpler alternative to ***SOAP*** and ***WSDL*** based Web services. REST defines a set of architectural principles by which you can design Web services that focus on a system's *resources*, including how resource states are addressed and transferred over ***HTTP*** by a wide range of clients written in different languages. Simply put, it is the architecture of the Web. REST has emerged as a predominant Web service design model. REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface design because it's a considerably simpler style to use.



- ***REST*** is preferable in problem domains that are query intense or that require exchange of large grain chunks of data. Basically, you would want to use RESTful web services for integration over the web.
- ***SOAP*** and ***WSDL*** based Web services ("Big" Web services) are preferable in areas that require asynchrony and various qualities of services. Finally, SOAP-based custom interfaces enable straightforward creation of new services based on choreography. Therefore, you will use big web services in enterprise application integration scenarios that have advanced quality of service (QoS) requirements.

Related tutorials:

- *https://en.wikipedia.org/wiki/Representational_state_transfer*
- *https://crunchify.com/soap-vs-rest-simple-object-access-protocol-vs-representational-state-transfer/*

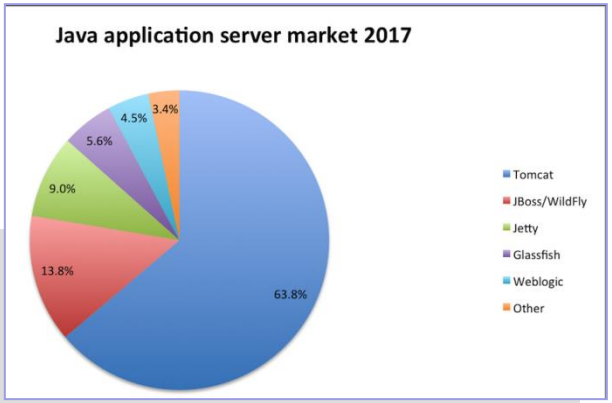
Demo

Web Services with Jetty and Java Servlets

The ***Jetty Web Server*** provides an HTTP server and Servlet container capable of serving static and dynamic content either from a standalone or embedded instantiations. (*http://www.eclipse.org/jetty/about.html*)

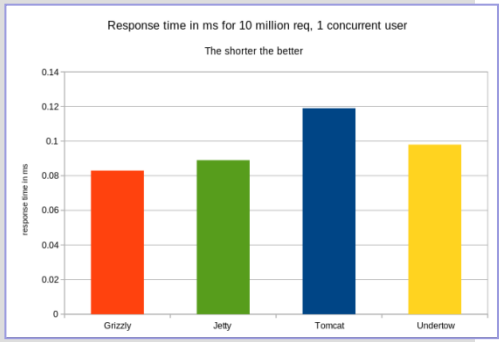
Servers comparison: *https://stackify.com/tomcat-vs-jetty-vs-glassfish-vs-wildfly/*

Postman helps you be extremely efficient while working with APIs. (*https://www.getpostman.com/apps*)




```
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletHandler;

public class RunServlets {
   public static void main(String[] args) throws Exception{
      ServletHandler handler = new ServletHandler();
//add all servlet to use to the handler, the second argument is the path (e.g. http://localhost:8080/searchPublication)
      handler.addServletWithMapping(SearchPublication.class, "/searchPublication");
      handler.addServletWithMapping(UserProfile.class, "/getProfile");
      handler.addServletWithMapping(CreateUserProfile.class, "/createProfile");
//Create a new Server, add the handler to it and start
      Server server = new Server(8080);
      server.setHandler(handler);
      server.start();
//this dumps a lot of debug output to the console.
      server.dumpStdErr();
      server.join();
   }
}
```

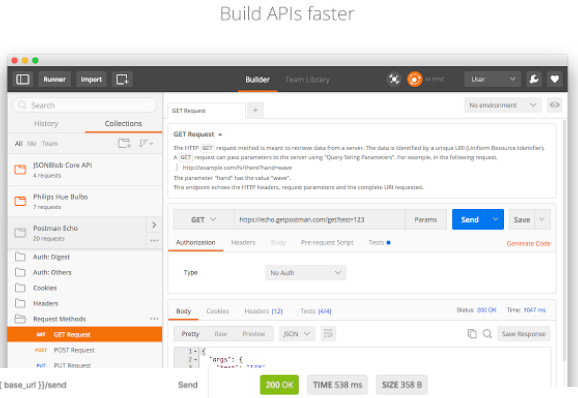


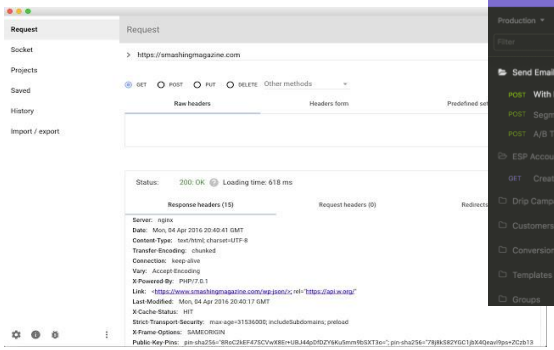
path	method	parameters	response	failures
/searchPublication	**GET**	searchString	response code 200 and JSON containing publication' information	response code 404 and JSON containing an error message
/createProfile	**POST**	*Data is sent in the POST body*	code 302 (or 303) and redirect to /getProfile?authorName=name	response code 400 and JSON containing an error message
/getProfile	**GET**	authorName	always fails	response code 501 and JSON message "is not implemented"

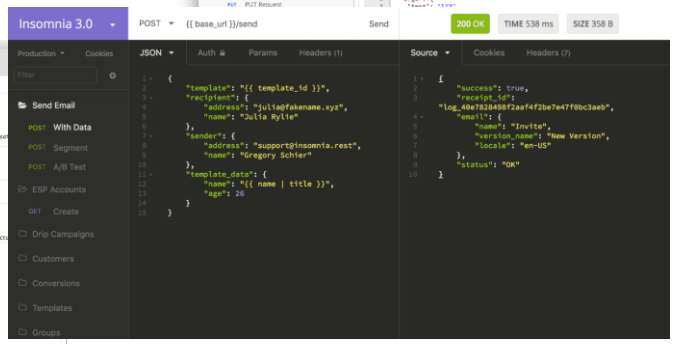
…see https://en.wikipedia.org/wiki/Post/Redirect/Get for redirection related issues.

REST Web Services

REST API Client *is the web developers helper program to create and test custom HTTP requests.*







Chrome

Postman: https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en

Advanced REST client: https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo

DHC: https://chrome.google.com/webstore/detail/dhc-rest-client/aejoelaoggembcahagimdiliamlcdmfm?hl=en

Firefox

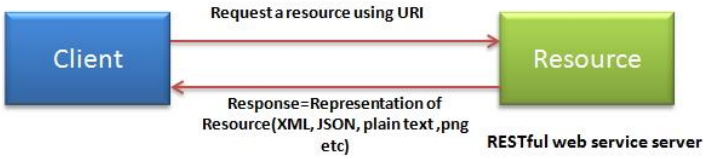
Firefox add-on: https://addons.mozilla.org/en-US/firefox/addon/restclient/

Insomnia *is a cross-platform application for organizing, running, and debugging HTTP requests (https://insomnia.rest/).*

RESTful Web Services

In the web services terms, *REpresentational State Transfer (REST)* is a stateless client-server architecture in which the web services are viewed as resources and can be identified by their URIs. Web service clients that want to use these resources access via globally defined set of remote methods that describe the action to be performed on the resource.

In the REST architecture style, *clients* and *servers* exchange representations of resources by using a standardized interface and protocol. REST isn't protocol specific, but when people talk about REST they usually mean REST over HTTP. The response from server is considered as the representation of the resources (that can be generated from one resource or more number of resources).



RESTFul web services are based on HTTP methods and the concept of REST. A RESTFul web service typically defines the *base URI* for the services, the supported *MIME-types* (XML, text, JSON, user-defined, ...) and the *set of operations* (POST, GET, PUT, DELETE) which are supported.

A concrete implementation of a *REST Web service* follows four basic design principles:

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JavaScript Object Notation (JSON), or both.

JAX-RS - Java API for RESTful Web Services, is a set of APIs to develop REST service. JAX-RS is part of the Java EE, and make developers to develop REST web application easily. There are two main implementation of *JAX-RS API*: *Jersey* and *RESTEasy*

Jersey is the open source, production quality, JAX-RS (JSR 311) Reference Implementation for building RESTful Web services. But, it is also more than the Reference Implementation. Jersey provides an API so that developers may extend Jersey to suit their needs.

Related practical tutorials:

- *https://www.ibm.com/developerworks/webservices/library/ws-restful/*
- *https://www.guru99.com/restful-web-services.html*
- *http://www.java2blog.com/2013/04/restful-web-service-tutorial.html*
- *http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html*
- *http://www.java2blog.com/2013/04/create-restful-web-servicesjax-rs-using.html*
- *http://www.java2blog.com/2016/04/spring-restful-web-services-crud-example.html*
- *http://www.tutorialspoint.com/restful/restful_first_application.htm*
- *http://www.vogella.com/tutorials/REST/article.html*

10 differences between SOAP and REST

No.	SOAP	REST
1)	SOAP is a protocol.	REST is an architectural style.
2)	SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.
3)	SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
4)	SOAP uses services interfaces to expose the business logic.	REST uses URI to expose business logic.
5)	JAX-WS is the java API for SOAP web services.	JAX-RS is the java API for RESTful web services.
6)	SOAP defines standards to be strictly followed.	REST does not define too much standards like SOAP.
7)	SOAP requires more bandwidth and resources than REST.	REST requires less bandwidth and resources than SOAP.
8)	SOAP defines its own security.	RESTful web services inherits security measures from the underlying transport.
9)	SOAP permits XML data format only.	REST permits different data format such as Plain text, HTML, XML, JSON etc.
10)	SOAP is less preferred than REST.	REST more preferred than SOAP.

http://www.javatpoint.com/soap-vs-rest-web-services
https://www.guru99.com/comparison-between-web-services.html

REST Web Services

*Stirred by **Web 2.0**, classical Web Services has significantly evolved with the proliferation of **Web APIs** – **RESTful services** (when confirm to the **REST** (Representational State Transfer) architectural style).*

*Web Apps became able to consume data as well as push/post data to other Apps through **Web API**.*

http://www.dropbox.com

HTML

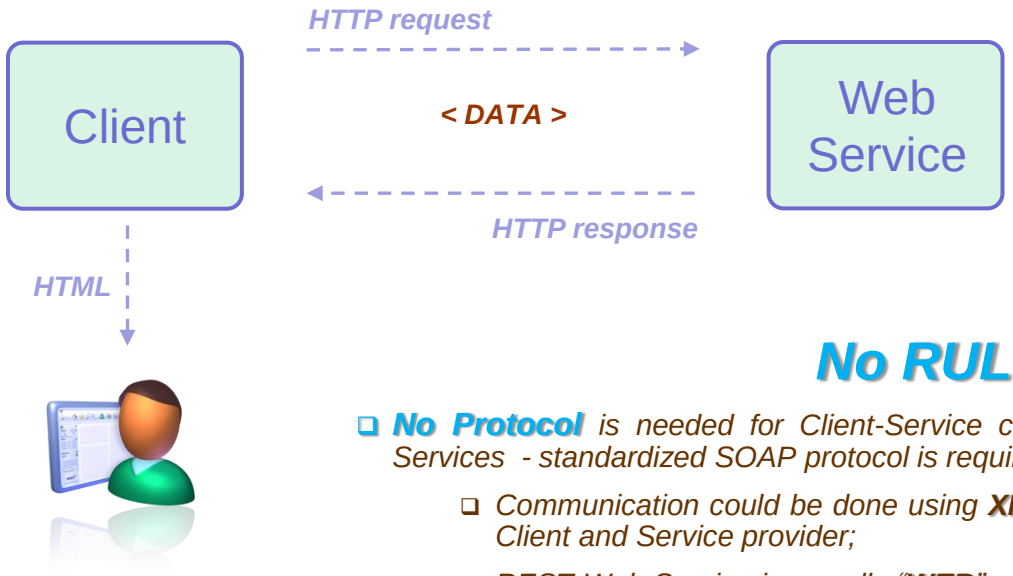
https://api.dropbox.com/

XML or JSON

***REST Web Services** are:*

- *Modern;*
- *Light weight;*
- *Use a lot of concepts behind HTTP*

REST Web Services



No RULES

- **No Protocol** is needed for Client-Service communication. (in case of SOAP Web Services - standardized SOAP protocol is required).
  - Communication could be done using **XML**, **JSON** or **Text** as long as it satisfies Client and Service provider;
  - REST Web Service is a really "**WEB**" service and all communication is done via HTTP using available in HTTP methods (GET, POST, PUT, DELETE, etc.);
- **No Service Definition** like WSDL in case of SOAP Web Services. Documentation of REST Web Service is usually done in a form of human readable page. The best implementation of a REST Web Service is an implementation that does not require any documentation!

The ONLY RULE – make it as easy as possible for use!

REST Web Services

Using standard HTTP methods, we ***get in the Response…***

- *Not only* ***URL*** *of the page to be displayed*
- ***Status Code*** *is very important in case of REST WS (e.g. 200 – Success, 500 – Server Error, 404- Not Found, etc.)*
- ***Metadata*** *in the Header of the message:*
  - *Content Type tells what format of data is used (e.g. text/xml, application/json, etc.)*
  - *Content Negotiation is used by client to specify preferable content format*

REST Web Services

Be stateless…

Client should send complete, independent requests, including within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component (service) to generate a response. (absence of state on the server removes the need to synchronize session data with an external application)

Stateless server-side components are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application.

REST Web Services

Cachebility…

Service *might generate responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a* ***Cache-Control*** *and* ***Last-Modified*** *(a date value) HTTP response header.*

Client *uses the* ***Cache-Control*** *response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the* ***Last-Modified*** *response header and sends back the date value in an* ***If-Modified-Since*** *header to ask the server if the resource has changed. This is called* ***Conditional GET****, and the two headers go hand in hand in that the server's response is a standard* ***304 code (Not Modified)*** *and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect by passing subsequent GET requests until the resource changes.*

Layered system…

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. ***Intermediary servers*** *may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.*

REST Web Services

Resource-based Address!!!

URI *should look like you are looking up for something that is already exists (like in case of static pages).*

www.myCV.org/members/OleksiyKhriyenko/

www.myCV.org/members/OleksiyKhriyenko/publications/

not like

an address to a service where you put some parameters to perform some actions…

www.myCV.org/getMembes

www.myCV.org/getMemberByName?memberName=OleksiyKhriyenko

www.myCV.org/getPublications?memberName=OleksiyKhriyenko

Use **nouns**, not **verbs** !!!

www.myCV.org/members/{memberName}

www.myCV.org/members/OleksiyKhriyenko/publications/{publicationId}

www.myCV.org/members/OleksiyKhriyenko/degrees/{degreeId}

REST Web Services

Resource relations…

*For example, **publication's** comments could be reached by following URI…*

Instance Resource URI

```
/members/{memberName}/publications/{publicationId}/comments/{commentId}
```

*… **dependence on what client knows** … If it is just a comment id, and client does not know id of publication and member id, then this information could not be retrieved.*

*If you need to access a collection of resources – use **Collection Resource URI***

Collection Resource URI

```
/members/{memberName}/publications/
```

```
/members/{memberName}/publications/{publicationId}/comments/
```

*They are **plural** – meaning that they represent a collection.*

To get all publications or comments irrespectively to particular person (member) or publication use…

Collection Resource URI

```
/publications/
```

```
/comments/
```

*… since there is **no rules** – it is up to you how to design URIs of your WS.*

REST Web Services

Filtering results…

*To filter the results, you may simply provide **query parameters**…*

```
/members/{memberName}/publications?offset=10&limit=15
```

Starts from publication number 10 and returns max15 following publications.

```
/members/{memberName}/publications?year=2018&type=journal
```

Returns only journal publications published in year 2018

Extra guidelines for URI structure…

- *Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs.*
- *Keep everything lowercase.*
- *Avoid using spaces Instead use hyphens (-) or underscores (_).*
- *Avoid query strings as much as you can.*
- *Instead of using the 404 Not Found code if the request URI is for a partial path, always provide a default page or resource as a response.*

URIs should also be static so that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking. It's also important that the relationship between resources that's encoded in the URIs remains independent of the way the relationships are represented where they are stored.

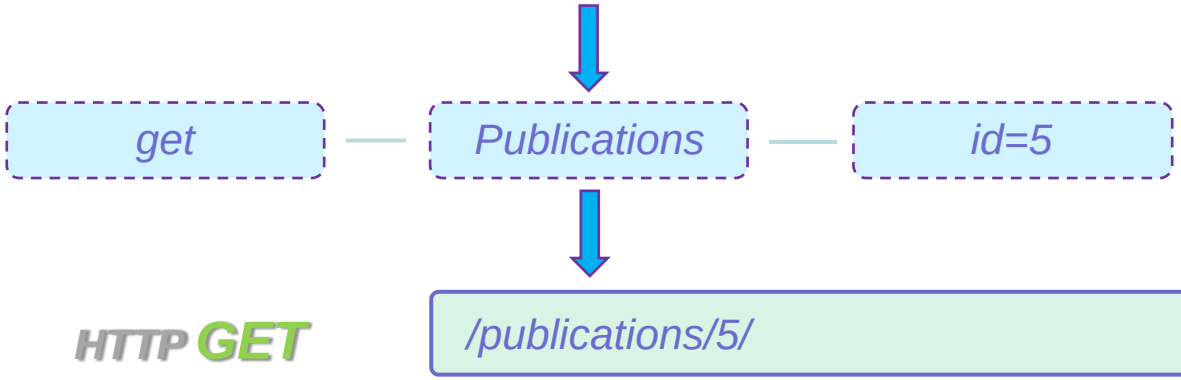
REST Web Services

Operation via HTTP Methods…

Get publication number 5…

Ordinary WS

/getPublication?id=5



Similarly …

/updateDegree?id=3	→	HTTP PUT	/degrees/3/
/deleteComment?id=7	→	HTTP DELETE	/comments/7/

REST Web Services

Operation via HTTP Methods…

HTTP GET `/publications/{publicationId}`

Get a publication under <id>…

Read-only method

Idempotent – is safely repeatable since does not make any changes

---

HTTP PUT `/publications/{publicationId}`

with new content in the message

Update/change a publication under <id>…

Write method

Idempotent – is repeatable since updates the same resource

HTTP DELETE `/publications/{publicationId}`

Delete a publication under <id>…

Write method

Idempotent – is repeatable since ones deleted there is nothing to delete anymore

---

HTTP POST `/publications/`

with correspondent publication in the message

since we do not know the publication Id, we just refer to the Collection URI. The response will contain an Id of created publication

Create a new publication…

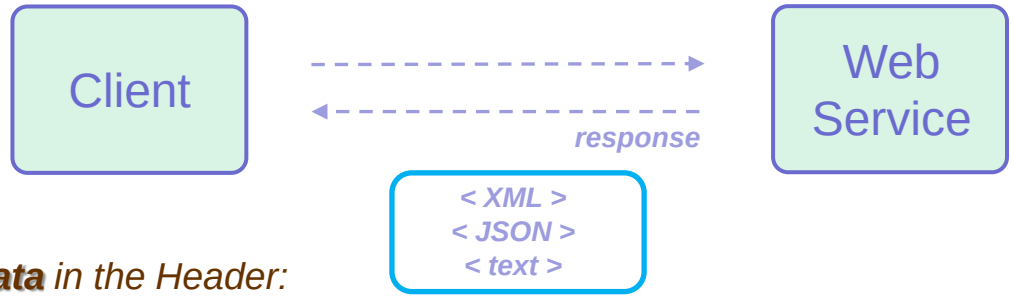
Write method

Non idempotent – is not repeatable since it will create every time some new resource

REST Web Services

REST Response…

*… is a **representation** of a resource. It could have several representations (e.g. XML, JSON, text, etc.)*



*… contains **metadata** in the Header:*

- *Status Code*
- *Message length*
- *Date*
- *Content Type*
- *Etc.*

Status Codes

- *"200 OK"*
- *"201 Created"*
- *"204 No Content"*
- *"302 Found"*
- *"304 Not modified"*
- *"307 Temporary Redirect"*
- *"400 Bad Request"*
- *"401 Unauthorized"*
- *"403 Forbidden"*
- *"404 Not Found"*
- *"415 Unsupported Media Type"*
- *"500 Internal Service Error"*

5 classes of codes:

- *1xx – Informational code*
- *2xx – Success code*
- *3xx – Redirection code*
- *4xx – Client Error code*
- *5xx – Service Error code*

*It is very good to **use Status Codes** as a service developer!!!*

REST Web Services

HATEOAS… *Hypermedia As The Engine Of Application State*

*A REST client **needs no prior knowledge** about how to interact with any particular application or server beyond a generic understanding of hypermedia. A REST client enters a REST application through a simple fixed URL. All future actions (the client may take) are discovered within resource representations returned from the server.*

Provide further guidance in the response!!!

```
{
    "id" : "10",
    "title"  : "Publication 123",
    "mainAuthor" : "me",
    "published" : "…",
    "co-Authors" : […]
}
```

Could be extended with some relevant data as links to the publication's comments and link to a profile of the main author…

```
{
    "id" : "10",
    "title"  : "Publication 123",
    "mainAuthor" : "me",
    "published" : "…",
    "co-Authors" : […],
    "commentsUri" : "…",
    "mainAuthorProfileUri" : "…"
}
```

REST Web Services

HATEOAS… *Hypermedia As The Engine Of Application State*

- *To separate extra relevant data from actual properties of a resource use concept of **Links**.*
- *Use "**href**" and "**rel**" attributes to specify URI of a linked document and relationship between the linked and original documents.*

```
{   "id" : "10",
    "title"  : "Publication 123",
    "mainAuthor" : "me",
    "published" : "…",
    "co-Authors" : […],
    "links" : [  {   "href" : "/publications/10",
                     "rel" : "self"   },
                 {   "href" : "/publications/10/comments",
                     "rel" : "comments"   },
                 {   "href" : "/profiles/3",
                     "rel" : "mainAuthorProfile"   }
             ]
}
```

*… client just needs to understand meaning of "**rel**" attribute!!!*

REST Web Services

Richardson Maturity Model

by Leonardo Richardson

Not RESTful — 0 ..... 1 ..... 2 ..... 3 — Fully RESTful

Level 0 *Not RESTful API (SOAP Web Service): has one URI, request body contains operation, everything in XML and no HTTP concepts are used.*

Level 1 *Introduce Resource-based URI, but still operations are contained in the request message.*

Level 2 *Introduce HTTP Methods to do different operations. Use of Status Codes.*

Level 3 *Introduce HATEOAS implementation.*

Task 2